March 1989

A Simple Semantics for ML Polymorphism

Atsushi Ohori
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-89-21.

A Simple Semantics for ML Polymorphism

Abstract

We give a framework for denotational semantics for the polymorphic "core" of the programming language ML. This framework requires no more semantic material than what is needed for modeling the *simple* type discipline. In our view, the terms of ML are pairs consisting of a raw (untyped) lambda term and a type-scheme that ML's type inference system can derive for the raw term. We interpret type-schemes as sets of simple types. Then, given *any* model M of the simply typed lambda calculus, the meaning of an ML term will be a set of pairs, each consisting of a simple type τ and an element of M of type τ.

Hence, there is no need to interpret all raw terms, as was done in Milner's original semantic framework. In comparison to Mitchell and Harper's analysis, we avoid having to provide a very large type universe in which generic type-schemes are interpreted. Also, we show how to give meaning to ML terms rather than to derivations in the ML type inference system (which can be several for the same term).

We give an axiomatization for the equational theory that corresponds to our semantic framework and prove the analogs of the compeleteness theorems that Friedman proved for the simply typed lambda calculus. The framework can be extended to languages with constants, type constructors and recursive types (via regular trees). For the extended language, we prove a theorem that allows the transfer of certain full abstraction results from languages based on the typed lambda calculus to ML-like languages.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-89-21.

A Simple Semantics For ML Polymorphism

MS-CIS-89-21
LOGIC & COMPUTATION 05

Atsushi Ohori

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

March 1989

ACKNOWLEDGMENTS:
This research was supported in part by grants NSF IRI86-10617, NSF MCS-8219196-CER, US Army grants DAA29-84-K-0061, DAA29-84-9-0027 and by funding from AT&T Telecommunications Program. Appeared in *Proceedings of Fourth ACM/IFIP Conference on Functional Programming Languages and Computer Architecture, London, England, ACM Press, September 1989.*

A Simple Semantics for ML Polymorphism*

Atsushi Ohori

Department of Computer and Information Science,
University of Pennsylvania,
200 South 33rd Street
Philadelphia, PA 19104-6389

Electric mail address : ohori@linc.cis.upenn.edu

Abstract

We give a framework for denotational semantics for the polymorphic "core" of the programming language ML. This framework requires no more semantic material than what is needed for modeling the *simple* type discipline. In our view, the terms of ML are pairs consisting of a raw (untyped) lambda term and a type-scheme that ML's type inference system can derive for the raw term. We interpret type-schemes as sets of simple types. Then, given *any* model $\mathcal{M}$ of the simply typed lambda calculus, the meaning of an ML term will be a set of pairs, each consisting of a simple type τ and an element of $\mathcal{M}$ of type τ.

Hence, there is no need to interpret all raw terms, as was done in Milner's original semantic framework. In comparison to Mitchell and Harper's analysis, we avoid having to provide a very large type universe in which generic type-schemes are interpreted. Also, we show how to give meaning to ML terms rather than to derivations in the ML type inference system (which can be several for the same term).

We give an axiomatization for the equational theory that corresponds to our semantic framework and prove the analogs of the compeleteness theorems that Friedman proved for the simply typed lambda calculus. The framework can be extended to languages with constants, type constructors and recursive types (via regular trees). For the extended language, we prove a theorem that allows the transfer of certain full abstraction results from languages based on the typed lambda calculus to ML-like languages.

1 Introduction

ML is a strongly typed programming language sharing with other typed languages the property that the type correctness of a program is completely checked by static analysis of the program – usually done at compile time. Among other strongly typed languages, one feature that distinguishes ML is its *implicit* type system. Unlike explicitly-typed languages such as Algol and the typed lambda calculus, ML does not require type specifications of bound variables. The syntax of ML programs is therefore same as that of untyped terms (raw

*This research was supported in part by grants NSF IRI86-10617, ARO DAA6-29-84-k-0061, and by OKI Electric Industry Co., Japan.

terms). However, not all raw terms correspond to legal ML programs. A term of ML is an associations of raw term and a type-scheme determined by a proof system often called a *type inference system.* Moreover, for any given raw term, the type system can *infer* its *most general* (or *principal*) type-scheme representing the set of all possible types of the raw term. For example, a type system can infer the type-scheme $t \to t$ for the raw term $\lambda x.\, x$, where t is a type variable representing arbitrary types. The above type-scheme correctly represents the set of all possible types of the raw term $\lambda x.\, x$. Through this type inference mechanism, ML attains the flexibility and convenience of untyped languages without sacrificing the desired feature of complete static type-checking. In the above example, the term $\lambda x.x$ can safely be used as a term of any type of the form $\tau \to \tau$. Combining the type inference with the binding mechanism of *let*-expressions, ML also realizes a form of *polymorphism* without using type abstraction or type application. In the body e of **let** $id = \lambda x.\, x$ **in** e, each occurrence of id can be used as an identity function of a different type.

There are two major existing approaches to explain ML type system — the one by Milner [Mil78] (extended by MacQueen, Plotkin and Sethi [MPS86]) based on a semantics of an untyped language and the other by Mitchell and Harper [MH88] based on an explicitly-typed language using Damas and Milner's type inference system [DM82]. As we shall suggest in this paper, however, neither of them properly explain the behavior of ML programs. Because of the implicit type system, ML behaves differently from both untyped languages and explicitly-typed languages. In order to understand ML, we need to develop a framework for denotational semantics and equational theories that give precise account for ML's implicit type system. The goal of this paper is to propose such a framework. In the rest of this section, we review the two existing approaches and outline our approach.

1.1 Milner's original semantics

In [Mil78], Milner proposed a semantic framework for ML based on a semantics of an *untyped* language. He defined the following two classes of types:

$$\begin{aligned} \tau &::= b \mid \tau \to \tau \\ \rho &::= t \mid b \mid \rho \to \rho \end{aligned}$$

where b stands for base types and t stands for type variables. Here we call them *types* (ranged over by τ) and *type-schemes* (ranged over by ρ) respectively. Type-schemes containing type variables represent all their substitution instances and correspond to polymorphic types. A type-scheme ρ_1 is *more general* than ρ_2 if ρ_2 is a substitution instance of ρ_1. He then gave the algorithm $\mathcal{W}$ that infers a most general type-scheme for the following raw terms:

$$e ::= x \mid (\lambda x.e) \mid (e\ e) \mid \mathit{if}\ e\ \mathit{then}\ e\ \mathit{else}\ e \mid \mathit{fix}\ x\ e \mid \textbf{let}\ x = e\ \textbf{in}\ e$$

where x stands for variables and *fix* x e stands for the least fixed point of $\lambda x.\, e$.

For this language, he proposed a semantic interpretation for the typing judgement $e : \rho$ as the set-membership relation between the denotation of e and the denotation of ρ and showed that the type inference algorithm $\mathcal{W}$ is sound under this interpretation. A denotation of a raw term is defined as an element of a

(universal) domain of untyped expressions given by the following domain equation:

$$V = B_1 + \cdots + B_n + [V \to V] + \{wrong\}$$

where $B_1, \ldots, B_n$ are basic domains and *wrong* represents run-time error. The denotation of types are inductively defined as subsets of V. The denotation of a type-scheme is defined as the intersection of the denotations of all its instance types. This semantics was extended to recursive types by MacQueen, Plotkin and Sethi [MPS86]. (See also [Hin83, Cop84] for related studies.)

This semantics explains the polymorphic nature of ML programs and verifies that ML typing discipline prevents all run-time type errors. However, this semantics does not fit the behavior of ML programs. As an example, consider the following two raw terms e_1 and e_2 with their type-schemes:

$$\begin{aligned} e_1 &\equiv \lambda x \lambda y.\, y \;:\; t_1 \to t_2 \to t_2 \\ e_2 &\equiv \lambda x \lambda y.\, (\lambda z \lambda w.\, w)(xy)y \;:\; (t_1 \to t_2) \to t_1 \to t_1 \end{aligned}$$

where parentheses are omitted assuming left association of applications. It is easily checked that the two raw terms have the same meaning under Milner's semantics. Indeed, if we were to ignore their type-schemes and regard them as terms in the untyped lambda calculus, then they would be β-convertible to each other and would be regarded as equal terms. However, ML is apparently a typed language, and as terms of ML, these two behave quite differently. For example, the term $((e_1\ 1)\ 2)$ is evaluated to 2 but $((e_2\ 1)\ 2)$ is not even a legal term and ML compiler reports a type error. This is one of the most noticeable difference between meanings of terms and should be distinguished by any semantics. From this example, we can also see that the equality on ML programs is different from the equality on terms in the untyped lambda calculus. For this reason, Milner's semantics seems to provide little help in reasoning about the behavior of ML programs.

1.2 Damas-Milner type inference system and Mitchell-Harper's analysis

Damas and Milner presented a proof system for typing judgements of ML [DM82]. They redefined the set of types of ML as the following two classes:

$$\begin{aligned} \rho &::= t \mid b \mid \rho \to \rho \\ \sigma &::= \rho \mid \forall t.\, \sigma \end{aligned}$$

The class represented by ρ is same as Milner's type-schemes. We call the class of types represented by σ *generic type-schemes*. t in $\forall t.\, \sigma$ is a bound type variable analogous to a bound variable in lambda terms. We write $\sigma[t_1 := \sigma_1, \ldots, t_n := \sigma_n]$ for the generic type-scheme obtained from σ by simultaneously substituting t_i by σ_i with necessary renaming of bound type variables. A type $\sigma = \forall t_1 \cdots t_n.\, \rho$ is a *generic instance* of a type $\sigma' = \forall t'_1 \cdots t'_m.\, \rho'$ if each t'_j is not free in σ and $\rho' = \rho[t_1 := \rho_1, \ldots, t_n := \rho_n]$ for some non-generic type-schemes $\rho_1, \ldots, \rho_n$. A type σ is *more general* than σ', $\sigma' < \sigma$, if σ' is a generic instance of σ. A Damas-Milner *type assignment scheme* Γ is a function from a finite subset of variables to generic type-schemes. For a given function f, we write $f\{x_1 := v_1, \ldots, x_n := v_n\}$ for the function f' such that $dom(f') = dom(f) \cup \{x_1, \ldots, x_n\}$, $f'(y) = f(y)$ for any $y \neq x_i, 1 \leq i \leq n$ and $f'(x_i) = v_i, 1 \leq i \leq n$. A Damas-Milner *typing-scheme* is a formula

of the form $\Gamma \rhd e : \sigma$ that is derivable in the following proof system:

$$\text{(VAR)} \qquad \Gamma \rhd x : \sigma \qquad \text{if } \Gamma(x) = \sigma$$

$$\text{(GEN)} \qquad \frac{\Gamma \rhd e : \sigma}{\Gamma \rhd e : \forall t.\, \sigma} \qquad \text{if } t \text{ not free in } \Gamma$$

$$\text{(INST)} \qquad \frac{\Gamma \rhd e : \sigma}{\Gamma \rhd e : \sigma'} \qquad \text{if } \sigma' < \sigma$$

$$\text{(ABS)} \qquad \frac{\Gamma\{x := \rho_1\} \rhd e : \rho_2}{\Gamma \rhd \lambda x.\, e : \rho_1 \to \rho_2}$$

$$\text{(APP)} \qquad \frac{\Gamma \rhd e_1 : \rho_1 \to \rho_2 \qquad \Gamma \rhd e_2 : \rho_1}{\Gamma \rhd (e_1\ e_2) : \rho_2}$$

$$\text{(LET)} \qquad \frac{\Gamma \rhd e_1 : \sigma \qquad \Gamma\{x := \sigma\} \rhd e_2 : \rho}{\Gamma \rhd \textbf{ let } x = e_1 \textbf{ in } e_2 : \rho}$$

We write $\boldsymbol{DM} \vdash \Gamma \rhd e : \sigma$ if $\Gamma \rhd e : \sigma$ is derivable in the proof system. Terms of ML are then defined as typing-schemes.

Based on this derivation system, Mitchell and Harper proposed another framework to explain implicit type system of ML [MH88]. In what follows, we shall only discuss their analysis of the "core" of ML. However, it should be mentioned that their approach also provides an elegant treatment of Standard ML's modules [HMM86].

They defined an explicitly-typed language, called Core-XML. The set of types of Core-XML is same as those in Damas-Milner system. The set of *un-checked* pre-terms of Core-XML is given by the following abstract syntax:

$$M ::= x \mid (M\ M) \mid (\lambda x : \rho.\, M) \mid (M\ \rho) \mid (\lambda t.\, M) \mid \textbf{ let } x : \sigma = M \textbf{ in } M$$

where $(M\ \rho)$ is a type application and $(\lambda t.\, M)$ is a type abstraction. Type-checking rules for Core-XML are given as follows:

$$\text{(VAR)} \qquad \Gamma \rhd x : \sigma \qquad \text{if } \Gamma(x) = \sigma$$

$$\text{(TABS)} \qquad \frac{\Gamma \rhd M : \sigma}{\Gamma \rhd (\lambda t.\, M) : \forall t.\, \sigma} \qquad \text{if } t \text{ not free in } \Gamma$$

$$\text{(TAPP)} \qquad \frac{\Gamma \rhd M : \forall t.\, \sigma}{\Gamma \rhd (M\ \rho) : \sigma[t := \rho]}$$

$$\text{(ABS)} \qquad \frac{\Gamma\{x := \rho_1\} \rhd M : \rho_2}{\Gamma \rhd (\lambda x : \rho_1.\, M) : \rho_1 \to \rho_2}$$

$$\text{(APP)} \qquad \frac{\Gamma \rhd M_1 : \rho_1 \to \rho_2 \qquad \Gamma \rhd M_2 : \rho_1}{\Gamma \rhd (M_1\ M_2) : \rho_2}$$

$$\text{(LET)} \qquad \frac{\Gamma \rhd M_1 : \sigma \qquad \Gamma\{x := \sigma\} \rhd M_2 : \rho}{\Gamma \rhd \textbf{ let } x : \sigma = M_1 \textbf{ in } M_2 : \rho}$$

We write $\boldsymbol{MH} \vdash \Gamma \rhd M : \sigma$ if $\Gamma \rhd M : \sigma$ is derivable from the above typing rules. Terms of Core-XML are typing-schemes that are derivable in the above system. They showed the following relationships between Core-XML and Damas-Milner system. Define the *type erasure* of a pre-term M, $erase(M)$, as follows:

$$\begin{aligned}
erase(x) &= x \\
erase((M_1\ M_2)) &= (erase(M_1)\ erase(M_2)) \\
erase((\lambda x : \rho.\ M)) &= (\lambda x.\, erase(M)) \\
erase((\lambda t.\ M)) &= erase(M) \\
erase((M\ \rho)) &= erase(M) \\
erase(\ \mathbf{let}\ x : \sigma = M_1\ \mathbf{in}\ M_2) &= \ \mathbf{let}\ x = erase(M_1)\ \mathbf{in}\ erase(M_2)
\end{aligned}$$

Theorem 1 (Mitchell-Harper) *If $\boldsymbol{MH} \vdash \Gamma \rhd M : \sigma$ then $\boldsymbol{DM} \vdash \Gamma \rhd erase(M) : \sigma$. If $\boldsymbol{DM} \vdash \Gamma \rhd e : \sigma$ then there exists a Core-XML pre-term M such that $erase(M) \equiv e$ and $\boldsymbol{MH} \vdash \Gamma \rhd M : \sigma$. Moreover, M can be computed effectively from a proof of $\Gamma \rhd e : \sigma$.* ∎

Based on this relationship, they concluded that Core-XML and Damas-Milner system are "equivalent" and regarded ML as a "convenient shorthand" for Core-XML.

If we could indeed regard ML terms as syntactic shorthands for Core-XML terms then equational theory and model theory could simply be those of Core-XML. However, we cannot simply regard ML terms as syntactic shorthands for Core-XML terms. "Syntactic shorthand" should mean a syntactic mapping from ML terms to Core-XML terms but the above relationship does not define nor imply such a mapping. The above results only established a correspondence between Core-XML terms and *derivations* of Damas-Milner terms. However, a Damas-Milner term in general has infinitely many distinct derivations. For example, consider the term:

$$\emptyset \rhd (\lambda x \lambda y.\, y)(\lambda x.\, x) : t \to t$$

This means that there are, in general, infinitely many distinct Core-XML terms that correspond to a given Damas-Milner term. One way to overcome this difficulty is to *choose* a particular Core-XML term among infinitely many choices. Such a choice seems possible if we assume a particular type inference algorithm. However we cannot regard any particular algorithm as a part of the essence of ML.

We also think that Damas-Milner system and the corresponding explicitly-typed language Core-XML are too strong to explain ML's type system. As argued by Milner in [Mil78], it is ML's unique feature and advantage that ML supports polymorphism without type abstraction and type application. Note that this account of ML only used non-generic type-schemes. As such a language, ML can be better understood without using generic type-schemes, whose semantics requires the construction of very large spaces.

1.3 A simple framework for ML polymorphism

From the above analyses, it appears that ML is different from both untyped languages and explicitly typed languages. In order to understand ML properly we need to develop a framework for semantics that account for ML's implicit type system. Such a semantics should be useful to reason about various properties of ML

programs including equality on programs and operational semantics. A strategy was already suggested in Mitchell-Harper approach. We can use an explicitly typed language as an intermediate language to define a semantics of ML. In this paper we propose a framework for semantics and equational theories of ML by extending Wand's analysis [Wan84], where ML terms are considered as shorthands for terms of the typed lambda calculus. Wand's approach, however, does not give semantics to terms whose type-schemes contain type variables.

We first define an inference system and semantics for ML *typings* (typing-schemes that do not contain type variables) and then "lift" them to general ML terms (i.e. typing-schemes). Analogous to the relationship between Damas-Milner system and Core-XML, derivations of typings correspond to terms of the simply typed lambda calculus. Here is the crucial point in the development of our semantic approach: we show that if two typed terms correspond to derivations of a same ML typing then they are β-convertible (theorem 7). This guarantees that any semantics of the simply typed lambda calculus, in which the rule (β) is sound, indeed yields a semantics of ML typings. We then regard a general ML term as a representation of a *set* of typings. An association of a raw term and a type-scheme is an ML term if the set of all its ground instances are ML typings. The denotation of an ML term is defined as the set of denotations of the typings indexed by types represented by its type-scheme. For example, we regard the denotation $[\![\emptyset \rhd \lambda x.\, x : t \to t]\!]$ as the set $\{(\tau \to \tau, [\![\lambda x : \tau.\, x]\!]) | \tau \in Type\}$.

Equational theories are defined not on raw terms but on typing-schemes. Informally, two typing-schemes are equal iff their type-schemes are equal and raw terms are convertible to each other. This definition correctly models the behavior of ML programs. Type-schemes determine the compile-time behavior of programs and raw terms determine their run-time behavior. We then prove the soundness and completeness of equational theories. This confirms that our notion of semantics precisely captures and justifies the informal intuition behind the behavior of ML programs.

Our semantic framework can be extended to languages with an arbitrary set of constants, an arbitrary set of type constructors and recursive types (via infinite regular trees). Our semantic framework can also be related to certain operational semantics. We show that if a semantics of the typed lambda calculus is fully abstract with respect to an operational semantics then the corresponding semantics of ML is fully abstract with respect to an operational semantics that satisfies certain reasonable properties in connection with the operational semantics of the typed lambda calculus. This results enables us to transfer various existing results for full abstraction of typed languages to ML-like languages. A limitation to this program is due to the fact that our interpretation needs the soundness of the (β) rule. Such models, of course, while good for "call-by-name" evaluation, are not computationally adequate for the usual "call-by-value" evaluation of ML programs. Thus, this result seems helpful only for "lazy" ML-like languages such as Mirand [Tur85]

2 The Language Core-ML

We first present our framework for the set of pure raw terms, the same set analyzed in [DM82, MH88]. We call the pure language Core-ML. Later in section 5 we extend our frameworks to a language allowing constants, arbitrary set of type constructors and recursive types (infinite types).

2.1 Raw terms, types and type-schemes

We assume that we are given a countably infinite set of variables Var (ranged over by x). The set of *raw terms* of Core-ML (ranged over by e) is defined by the following abstract syntax:

$$e ::= x \mid (e\ e) \mid \lambda x.\, e \mid \ \mathbf{let}\ x = e\ \mathbf{in}\ e$$

We write $e[x_1 := e_1, \ldots, x_n := e_n]$ for the raw term obtained from e by simultaneously replacing $x_1, \ldots, x_n$ by $e_1, \ldots, e_n$ with necessary bound variable renaming. The intended meaning of $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$ is to *bind* x to e_1 in e_2 and to denote operationally the expression $e_2[x := e_1]$. For a raw term e, the *let expansion* e, $letexpd(e)$, is the raw term without *let*-expression obtained from e by repeatedly replacing the outmost subterm of the form $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$ by $e_2[x := e_1]$. For any raw term e, $letexpd(e)$ always exists.

The set of *types*, $Types$ ranged over by τ, is given by the following abstract syntax:

$$\tau ::= b \mid \tau \rightarrow \tau$$

where b stands for base types. We assume that there is a given set $Tvar$ of countably infinite type variables (ranged over by t). The set of *type-schemes*, $Tscheme$ ranged over by ρ, is given by the following abstract syntax:

$$\rho ::= t \mid b \mid \rho \rightarrow \rho$$

A *substitution* θ is a function from $Tvar$ to $Tscheme$ such that $\theta(t) \neq t$ for only finitely many t. We identify θ with its extension to $Tscheme$ (and any other tree structures that contain type-schemes). A type-scheme ρ is an *instance* of a type-scheme ρ' if there is a substitution θ such that $\theta(\rho') = \rho$. If ρ is a type then it is a *ground* instance.

2.2 Typings, typing-schemes and terms of Core-ML

A *type assignment* $\mathcal{A}$ is a function from a finite subset of Var to $Types$. A *typing* is a formula of the form $\mathcal{A} \rhd e : \tau$ that is derivable in the following proof system:

$$\text{(VAR)} \qquad \mathcal{A} \rhd x : \tau \qquad \text{if } \mathcal{A}(x) = \tau$$

$$\text{(APP)} \qquad \frac{\mathcal{A} \rhd e_1 : \tau_1 \rightarrow \tau_2 \qquad \mathcal{A} \rhd e_2 : \tau_1}{\mathcal{A} \rhd (e_1\ e_2) : \tau_2}$$

$$\text{(ABS)} \qquad \frac{\mathcal{A}\{x := \tau_1\} \rhd e : \tau_2}{\mathcal{A} \rhd \lambda x.\, e : \tau_1 \rightarrow \tau_2}$$

$$\text{(LET)} \qquad \frac{\mathcal{A} \rhd e_1[x := e_2] : \tau \qquad \mathcal{A} \rhd e_2 : \tau' \text{ for some } \tau'}{\mathcal{A} \rhd \ \mathbf{let}\ x = e_2\ \mathbf{in}\ e_1 : \tau}$$

We write $\boldsymbol{ML} \vdash \mathcal{A} \rhd e : \tau$ if $\mathcal{A} \rhd e : \tau$ is derivable in the above proof system. A *derivation* Δ of $\mathcal{A} \rhd e : \tau$ is a proof tree for $\mathcal{A} \rhd e : \tau$ in the above proof system.

A *type assignment scheme* Σ is a function from a finite subset of Var to $Tscheme$. A *typing-scheme* is a formula of the form $\Sigma \rhd e : \rho$ whose ground instances are all typings, i.e. for any substitution θ if $\theta(\Sigma)$ is a

type assignment and $\theta(\rho)$ is a type then $\boldsymbol{ML} \vdash \theta(\Sigma) \rhd e : \theta(\rho)$. We write $\boldsymbol{ML} \vdash \Sigma \rhd e : \rho$ if $\Sigma \rhd e : \rho$ is a typing-scheme. A typing-scheme $\Sigma_1 \rhd e : \rho_1$ is *more general than* a typing-scheme $\Sigma_2 \rhd e : \rho_2$, write $\Sigma_2 \rhd e : \rho_2 < \Sigma_1 \rhd e : \rho_1$, if there is a substitution θ such that $\Sigma_2 \upharpoonright^{dom(\Sigma_1)} = \theta(\Sigma_1)$ and $\rho_2 = \theta(\rho_1)$, where $f \upharpoonright^X$ denotes the function restriction of f on X. Note that more general also means less entries in a type assignment scheme. A typing-scheme $\Sigma \rhd e : \rho$ is *principal* if $\Sigma' \rhd e : \rho' < \Sigma \rhd e : \rho$ for any typing-scheme $\Sigma' \rhd e : \rho'$. The following property is an immediate consequence of the definition:

Proposition 1 *If $\Sigma \rhd e : \rho$ is a principal typing-scheme then $\{\mathcal{A} \rhd e : \tau | \mathcal{A} \rhd e : \tau < \Sigma \rhd e : \rho\} = \{\mathcal{A} \rhd e : \tau | \boldsymbol{ML} \vdash \mathcal{A} \rhd e : \tau\}$.* ∎

This means that a principal typing-scheme represents the set of all provable typings. In what follows, we regard typing-schemes as representatives of equivalence classes under the preorder $<$, i.e. equivalence classes induced by renaming of type variables (without "collapsing" distinct variables).

We now define terms of ML as (not necessarily principal) typing-schemes. Non principal typing-schemes correspond to programs with (partial) type specifications which are supported in ML and can be easily added to our definition. A term containing type variables corresponds to a polymorphic program in ML. A raw term e in $\Sigma \rhd e : \rho$ represents the computational contents of the term and determines its run-time behavior. The pair (Σ, ρ) represents the typing contexts in which e is meaningful and determines the compile-time behavior of the term.

One important property of Core-ML is the decidability of type inference problem:

Theorem 2 *If a raw term e has a typing-scheme then it has a principal typing-scheme. Moreover, there is an algorithm which, given a raw term, computes a principal typing-scheme if one exists otherwise reports failure.*

Proof This is a simple extension of Hindley's result of principal typing-schemes for untyped lambda term [Hin69] by using the following property: $\boldsymbol{ML} \vdash \mathcal{A} \rhd$ **let** $x = e_1$ **in** $e_2 : \tau$ iff $\boldsymbol{ML} \vdash \mathcal{A} \rhd e_2[x := e_1] : \tau$ and $\boldsymbol{ML} \vdash \mathcal{A} \rhd e_1 : \tau'$ for some τ'. ∎

Since the relation $\Sigma_1 \rhd e : \rho_1 < \Sigma_2 \rhd e : \rho_2$ is decidable, this theorem also establishes the decidability of the provability $\boldsymbol{ML} \vdash \Sigma \rhd e : \rho$.

This typing derivation system is significantly simpler than that of Damas-Milner system. In particular, it does not involve generic type-schemes. However, for closed terms, they are essentially equivalent in the sense of the following two theorems (theorem 3 and 4):

Theorem 3 *For a closed raw term e, if $\boldsymbol{DM} \vdash \Gamma \rhd e : \sigma$ then there is some Σ such that $\boldsymbol{ML} \vdash \Sigma \rhd e : \rho$ where ρ is the most general non-generic type-scheme such that $\rho < \sigma$, i.e ρ is the type-scheme obtained from σ by deleting all prefixes of the form $\forall t$.*

Proof The proof uses the following lemma. We only show outline of their proofs.

Lemma 1 *If $\boldsymbol{DM} \vdash \Gamma \rhd e : \sigma$ then it has a derivation such that all applications of the rule* (INST) *are immediately preceded by an instance of the axiom scheme* (VAR).

Proof By induction on the structure of e. ∎

Lemma 2 *For raw terms* e_1, e_2, $\boldsymbol{DM} \vdash \Gamma \rhd$ **let** $x = e_1$ **in** $e_2 : \rho$ *iff* $\boldsymbol{DM} \vdash \Gamma \rhd e_1 : \sigma$ *for some* σ *and* $\boldsymbol{DM} \vdash \Gamma \rhd e_2[x := e_1] : \rho$.

Proof This is proved by lemma 1, the existence of most general typing-schemes in Damas-Milner system and the following property: if $\boldsymbol{DM} \vdash \Gamma\{x := \rho_1\} \rhd e_1 : \rho_2$ and $\boldsymbol{DM} \vdash \Gamma \rhd e_2 : \rho_1$ then $\boldsymbol{DM} \vdash \Gamma \rhd e_1[x := e_2] : \rho_2$. ∎

We now prove the theorem. Suppose $\boldsymbol{DM} \vdash \Gamma \rhd e : \sigma$. Since e is a closed term, $\boldsymbol{DM} \vdash \emptyset \rhd e : \sigma$. By the rule (INST), $\boldsymbol{DM} \vdash \emptyset \rhd e : \rho$ where ρ is the most general non-generic type-scheme such that $\rho < \sigma$. By the property of $letexpd(e)$ and lemma 2, $\boldsymbol{DM} \vdash \emptyset \rhd letexpd(e) : \rho$. By lemma 1 and the fact that $letexpd(e)$ does not contain *let*-expression, $\emptyset \rhd letexpd(e) : \rho$ has a derivation Δ such that it does not contain applications of (GEN) or (INST). This means that any ground instance of Δ is a derivation in $\boldsymbol{ML} \vdash$. Therefore $\boldsymbol{ML} \vdash \mathcal{A} \rhd letexpd(e) : \tau$ for any ground instance $(\mathcal{A}, \tau)$ of (Σ, ρ). But in $\boldsymbol{ML} \vdash$, it is easily shown by definition that $\boldsymbol{ML} \vdash \mathcal{A} \rhd e : \tau$ iff $\boldsymbol{ML} \vdash \mathcal{A} \rhd letexpd(e) : \tau$. Therefore $\boldsymbol{ML} \vdash \emptyset \rhd e : \tau$ for any ground instance $(\mathcal{A}, \tau)$ of (Σ, ρ). Hence $\boldsymbol{ML} \vdash \emptyset \rhd e : \rho$. ∎

Theorem 4 *For any raw term* e, *if* $\boldsymbol{ML} \vdash \Sigma \rhd e : \rho$ *then there is some* Γ *such that* $\boldsymbol{DM} \vdash \Gamma \rhd e : \rho$.

Proof By induction on the structure of e using lemma 2. ∎

As we have demonstrated through theorem 2, 3, and 4, ML's syntactic properties are understood without using generic type-schemes. This correspond to our semantics which only requires the semantic space of the simply typed lambda calculus. However, our system suggests a potentially inefficient type inference algorithm. A straightforward implementation of an algorithm based on our derivation system would infer a typing-scheme of **let** $x = e_1$ **in** e_2 by inferring a typing-scheme of $e_2[x := e_1]$. This involves repeated inferences of a typing-scheme of e_1 because of multiple occurrences of x in e_2, which is clearly redundant. The extra typing rules for generic type-schemes in Damas-Milner system and the corresponding control structures of the algorithm $\mathcal{W}$ can be regarded as a mechanism to eliminate the redundancy and can be considered as implementation aspects of ML type inference.

2.3 Equational theories of Core-ML

A formula of an equational theory is a pair of terms having the same type assignment scheme and the same type-scheme. We write $\Sigma \rhd e_1 = e_2 : \rho$ for such a pair. An *ML-theory* consists of a given set of equations E_{ML} and the following set of rules: the axiom schemes $(\alpha), (\beta), (\eta)$, the inference rule scheme (ξ) obtained from respective rule schemes in the untyped lambda calculus by *tagging* Σ and ρ (and restricting axioms to the set of pairs of legal terms of ML), the set of rule schemes for usual equational reasoning (i.e. reflexivity, symmetry, transitivity and congruence), the following axiom scheme:

$$(let) \qquad \Sigma \rhd (\textbf{ let } x = e_1 \textbf{ in } e_2) = (e_2[x := e_1]) : \rho,$$

and the following inference rule scheme:

$$(\textit{thinning})\ \frac{\Sigma \rhd e_1 = e_2 : \rho}{\Sigma' \rhd e_1 = e_2 : \rho} \qquad \text{if } \Sigma \subseteq \Sigma' \text{ (as graphs).}$$

On a set of equations E_{ML}, we require the following closure properties:

1. if $\Sigma \rhd e_1 = e_2 : \rho \in E$ and (Σ', ρ') is an instance of (Σ, ρ) then $\Sigma' \rhd e_1 = e_2 : \rho' \in E$.
2. if $\Sigma_1 \rhd e_1 = e_2 : \rho_1 \in E$ and $\Sigma_2 \rhd e_1 = e_2 : \rho_2 \in E$ then there is some $\Sigma \rhd e_1 = e_2 : \rho \in E$ such that (Σ_1, ρ_1) and (Σ_2, ρ_2) are instances of (Σ, ρ).

We believe that all useful equations for ML satisfy the restrictions. We call a set of equation E_{ML} satisfying the above closure properties as a set of *ML-equations*. We write $E_{\mathrm{ML}} \vdash_{\mathrm{ML}} \Sigma \rhd e_1 = e_2 : \rho$ if $\Sigma \rhd e_1 = e_2 : \rho$ is derivable from the axioms and E_{ML} using the inference rules. A set of ML-equations E_{ML} determines the ML-theory $Th_{\mathrm{ML}}(E_{\mathrm{ML}})$. We sometimes regard $Th_{\mathrm{ML}}(E_{\mathrm{ML}})$ as the set of all equations that are provable by the theory. The theory $Th_{\mathrm{ML}}(\emptyset)$ corresponds to the equality on ML terms. We write $\Sigma \rhd e_1 =_{\mathrm{ML}} e_2 : \rho$ for $\emptyset \vdash_{\mathrm{ML}} \Sigma \rhd e_1 = e_2 : \rho$.

If we exclude the rule of symmetry from the set of rules, then we have the notion of *reductions*. We write $E_{\mathrm{ML}} \vdash_{\mathrm{ML}} \Sigma \rhd e_1 \twoheadrightarrow e_2 : \rho$ if $\Sigma \rhd e_1 : \rho$ is reducible to $\Sigma \rhd e_2 : \rho$ using E_{ML} and the set of rules. In particular, the empty set determines the $\beta\eta$-reducibility, for which we write $\mathcal{A} \rhd M_1 \twoheadrightarrow_{\mathrm{ML}} M_2 : \tau$. The notion of *normal forms* are defined accordingly.

3 Semantics of Core-ML

In this section, we first define the explicitly-typed language $T\Lambda$ that corresponds to derivations of Core-ML typings. We then define the semantics of Core-ML relative to a semantics of $T\Lambda$.

3.1 Explicitly-typed language $T\Lambda$ and its semantics

The set of types of $T\Lambda$ is exactly the set *Types* of types of ML. The set of un-checked pre-terms is given by the following abstract syntax:

$$M ::= x \mid (M\ M) \mid \lambda x : \tau.\, M$$

Type-checking rules for $T\Lambda$ are given as follows:

$$(\textsc{var}) \qquad \mathcal{A} \rhd x : \tau \qquad \text{if } \Gamma(x) = \tau$$

$$(\textsc{abs}) \qquad \frac{\mathcal{A}\{x := \tau_1\} \rhd M : \tau_2}{\mathcal{A} \rhd (\lambda x : \tau_1.\, M) : \tau_1 \to \tau_2}$$

$$(\textsc{app}) \qquad \frac{\mathcal{A} \rhd M_1 : \tau_1 \to \tau_2 \qquad \mathcal{A} \rhd M_2 : \tau_1}{\mathcal{A} \rhd (M_1\ M_2) : \tau_2}$$

We write $\boldsymbol{T\Lambda} \vdash \mathcal{A} \rhd M : \tau$ if $\mathcal{A} \rhd M : \tau$ is derivable from the above typing rules. The set of terms of $T\Lambda$ is the set of all derivable typings. $T\Lambda$ is clearly a representation of the simply typed lambda calculus, whose equational theory and model theory are well understood.

We write $\mathcal{A} \rhd M_1 = M_2 : \tau$ for an equation of $T\Lambda$. A *$T\Lambda$-theory* consists of a set $E_{T\Lambda}$ of equations and the following set of rules: the axiom schemes $(\alpha), (\beta), (\eta)$ and the inference rule scheme (ξ) of the simply typed lambda calculus, the set of rule schemes for usual equational reasoning and the following inference rule scheme:

$$(thinning)\ \frac{\mathcal{A} \rhd M_1 = M_2 : \tau}{\mathcal{A}' \rhd M_1 = M_2 : \tau}\ \text{if } \mathcal{A} \subseteq \mathcal{A}' \text{ (as graphs)}$$

We write $E_{T\Lambda} \vdash_{T\Lambda} \mathcal{A} \rhd M_1 = M_2 : \tau$ if $\mathcal{A} \rhd M_1 = M_2 : \tau$ is derivable from the axioms and $E_{T\Lambda}$ using the inference rules. The following notations and notions are defined parallel to Core-ML: $Th_{T\Lambda}(E_{T\Lambda})$, $\mathcal{A} \rhd M_1 =_{T\Lambda} M_2 : \tau$, the notion of reductions, $E_{T\Lambda} \vdash_{T\Lambda} \mathcal{A} \rhd M_1 \twoheadrightarrow M_2 : \tau$, and $\mathcal{A} \rhd M_1 \twoheadrightarrow_{T\Lambda} M_2 : \tau$.

Following [Fri73] we define the notion of *models* of $T\Lambda$ as follows: A *frame* is a pair $(\mathcal{D}, \bullet)$ where $\mathcal{D}$ is a set $\{D_\tau | \tau \in Types\}$ such that each D_τ is non-empty and $\bullet$ is a family of binary operations $\bullet_{\tau_1,\tau_2} : D_{\tau_1 \to \tau_2} \times D_{\tau_1} \to D_{\tau_2}$. A frame is *extensional* if

$$\forall \tau_1, \tau_2 \in Types, \forall f, g \in D_{\tau_1 \to \tau_2}.(\forall d \in D_{\tau_1}.f \bullet d = g \bullet d) \Longrightarrow f = g$$

Given a frame $\mathcal{F}$, a map $\phi : D_{\tau_1} \to D_{\tau_2}$ is *representable* if there is some $f \in D_{\tau_1 \to \tau_2}$ such that $\forall d \in D_{\tau_1}.\phi(d) = f \bullet d$ (f is a *representative* of ϕ). In an extensional frame, representatives are unique. For a frame $\mathcal{F} = (\mathcal{D}, \bullet)$ and a type assignment $\mathcal{A}$, a $\mathcal{FA}$-environment ε is a mapping from $dom(\mathcal{A})$ to $\bigcup \mathcal{D}$ such that $\varepsilon(x) \in D_{\mathcal{A}(x)}$. We write $Env^{\mathcal{F}}(\mathcal{A})$ for the set of all $\mathcal{FA}$-environments. A semantics of a term $\mathcal{A} \rhd M : \rho$ is a mapping from $Env^{\mathcal{F}}(\mathcal{A})$ to D_ρ. An extensional frame $\mathcal{M}$ is a *model* if there is a semantic mapping $[\![\,]\!]$ on terms of $T\Lambda$ satisfying the following equations. For any $\varepsilon \in Env^{\mathcal{M}}(\mathcal{A})$:

$$\begin{aligned}
[\![\mathcal{A} \rhd x : \tau]\!]\varepsilon &= \varepsilon(x) \\
[\![\mathcal{A} \rhd \lambda x : \tau_1.\, M : \tau_1 \to \tau_2]\!]\varepsilon &= \text{the representative of } \phi \text{ such that} \\
&\quad (\forall d \in D_{\tau_1})(\phi(d) = [\![\mathcal{A}\{x := \tau_1\} \rhd M : \tau_2]\!]\varepsilon\{x := d\}) \\
[\![\mathcal{A} \rhd (M\ N) : \tau]\!]\varepsilon &= [\![\mathcal{A} \rhd M : \tau_1 \to \tau]\!]\varepsilon \bullet [\![\mathcal{A} \rhd N : \tau_1]\!]\varepsilon
\end{aligned}$$

Note that for a given extensional frame, such a semantic mapping does not necessarily exist, but if one exists then it is unique. If $\mathcal{M}$ is a model, then we write $\mathcal{M}[\![\,]\!]$ for the unique semantic mapping.

An equation $\mathcal{A} \rhd M = N : \tau$ is *valid* in a model $\mathcal{M}$, write $\mathcal{M} \models_{T\Lambda} \mathcal{A} \rhd M = N : \tau$, if $\mathcal{M}[\![\mathcal{A} \rhd M : \tau]\!] = \mathcal{M}[\![\mathcal{A} \rhd M : \tau]\!]$. Let $\boldsymbol{Valid}^{T\Lambda}(\mathcal{M})$ be the set of all $T\Lambda$ equations that are valid in $\mathcal{M}$. Write $\mathcal{M} \models_{T\Lambda} F$ for $F \subseteq \boldsymbol{Valid}^{T\Lambda}(\mathcal{M})$. For $T\Lambda$ we have the following soundness and completeness of equational theories [Fri73]:

Theorem 5 (Friedman) *For any model $\mathcal{M}$ and any $T\Lambda$-theory $Th_{T\Lambda}(E_{T\Lambda})$, if $\mathcal{M} \models_{T\Lambda} E_{T\Lambda}$ then $Th_{T\Lambda}(E_{T\Lambda}) \subseteq \boldsymbol{Valid}^{T\Lambda}(\mathcal{M})$. For any $T\Lambda$-theory T, there exists a model $\mathcal{T}$ such that $\boldsymbol{Valid}^{T\Lambda}(\mathcal{T}) = T$.*

3.2 Relationship between $T\Lambda$ and derivations of Core-ML typings

Analogous to the relationship between Damas-Milner system and Core-XML, derivations of Core-ML typings correspond to terms of $T\Lambda$. Define a mapping *typedterm* on derivations of Core-ML typings as follows:

(1) If Δ is the one node derivation tree

$$\mathcal{A} \rhd x : \tau \qquad \text{(VAR)}$$

then $typedterm(\Delta) = x$.

(2) If Δ is the tree of the form

$$\frac{\Delta_1}{\mathcal{A} \rhd \lambda x.\, e_1 : \tau_1 \to \tau_2} \qquad \text{(ABS)}$$

then $typedterm(\Delta) = \lambda x : \tau_1.\, typedterm(\Delta_1)$.

(3) If Δ is the tree of the form

$$\frac{\Delta_1 \qquad \Delta_2}{\mathcal{A} \rhd (e_1\ e_2) : \tau} \qquad \text{(APP)}$$

then $typedterm(\Delta) = (typedterm(\Delta_1)\ typedterm(\Delta_2))$.

(5) If Δ is the tree of the form

$$\frac{\Delta_1}{\mathcal{A} \rhd \textbf{ let } x = e_1 \textbf{ in } e_2 : \tau} \qquad \text{(LET)}$$

then $typedterm(\Delta) = typedterm(\Delta_1)$.

The *type erasure* of a pre-term M, write $erase(M)$, is the raw term obtained from M by *erasing* all type specifications of the form ": τ" in all subterms of the form $\lambda x : \tau.M'$ in M. The following theorem corresponds to theorem 1:

Theorem 6 *If* $\boldsymbol{T\Lambda} \vdash \mathcal{A} \rhd M : \tau$ *then* $\boldsymbol{ML} \vdash \mathcal{A} \rhd erase(M) : \tau$ *and there is a derivation* Δ *for* $\mathcal{A} \rhd erase(M) : \tau$ *such that* $typedterm(\Delta) \equiv M$. *If* Δ *is a typing derivation for* $\mathcal{A} \rhd e : \tau$ *then* $letexpd(e) \equiv erase(typedterm(\Delta))$ *and* $\boldsymbol{T\Lambda} \vdash \mathcal{A} \rhd typedterm(\Delta) : \tau$.

Proof These properties are shown by inductions on the structures of M and e respectively. ∎

On this relationship we also have the following desired property:

Theorem 7 *If* Δ_1, Δ_2 *are typing derivations of a same typing* $\mathcal{A} \rhd e : \tau$ *then the following equation holds:*

$$\mathcal{A} \rhd typedterm(\Delta_1) =_{T\Lambda} typedterm(\Delta_2) : \tau.$$

Proof The proof uses the following lemmas:

Lemma 3 *Let* $\mathcal{A} \rhd M : \tau$ *and* $\mathcal{A} \rhd e : \tau$ *be respectively* $T\Lambda$ *term and Core-ML term such that* $erase(M) \equiv e$. *If* $\mathcal{A} \rhd M \twoheadrightarrow_{T\Lambda} M' : \tau$ *then there is* e' *such that* $erase(M') \equiv e'$ *and* $\mathcal{A} \rhd e \twoheadrightarrow_{\mathrm{ML}} e' : \tau$. *Conversely, if* $\mathcal{A} \rhd e_1 \twoheadrightarrow_{\mathrm{ML}} e_2 : \tau$ *then there is* M' *such that* $erase(M') \equiv e'$ *and* $\mathcal{A} \rhd M \twoheadrightarrow_{T\Lambda} M' : \tau$.

Proof This is proved by observing the following facts: (1) there is a one-one correspondence between the set of $\beta\eta$-redexes in M and the set of $\beta\eta$-redexes in e, (2) if $erase((\lambda x : \tau.\, M_1)\ M_2) \equiv ((\lambda x.e_1)\ e_2)$ then $erase(M_1[x := M_2]) \equiv e_1[x := e_2]$, and (3) if $erase(\lambda x : \tau.\, Mx) \equiv (\lambda x.ex)$ then $erase(M) \equiv e$. ∎

Note that this result, combined with the property of the reduction rule (*let*) and the connection between $T\Lambda$ terms and typing derivations of ML implies that if $T\Lambda$ has the strong normalization property then so does Core-ML, which was suggested in [HS86, remark 15.32]. Technical difficulty of treating bound variables mentioned in [HS86, remark 15.32] was overcome by our presentation of $T\Lambda$.

Lemma 4 *If two terms $\mathcal{A} \rhd M_1 : \tau$ and $\mathcal{A} \rhd M_2 : \tau$ are in β-normal form and $erase(M_1) \equiv erase(M_2)$ then $M_1 \equiv M_2$.*

Proof The proof is by induction on the structure of M_1. Bases are trivial. Induction step is by cases.

Case of $M_1 \equiv \lambda x : \tau_1.\, M_1'$: By the typing rules of $T\Lambda$, $\boldsymbol{T\Lambda} \vdash \mathcal{A}\{x := \tau_1\} \rhd M_1' : \tau_2$ for some τ_2 and $\tau = \tau_1 \to \tau_2$. Since $erase(M_1) \equiv erase(M_2)$, M_2 must be of the form $\lambda x : \tau_1'.\, M_2'$ such that $erase(M_1') \equiv erase(M_2')$. By the typing rules for $T\Lambda$, $\boldsymbol{T\Lambda} \vdash \mathcal{A}\{x := \tau_1'\} \rhd M_2 : \tau_2'$ for some τ_2' and $\tau = \tau_1' \to \tau_2'$. Therefore $\tau_1 = \tau_1', \tau_2 = \tau_2'$. By definition, $\mathcal{A}\{x := \tau_1\} \rhd M_1' : \tau_2$ and $\mathcal{A}\{x := \tau_1\} \rhd M_2' : \tau_2$ must be also in β-normal form. Then by induction hypothesis, $M_1' \equiv M_2'$. This implies $M_1 \equiv M_2$.

Case of $M_1 \equiv (\cdots(x\ M_1^1)\cdots M_1^n)$: Define $N_1^n \equiv x, N_1^i \equiv (\cdots(x\ M_1^1)\cdots M_1^{n-i}), 0 \le i \le n-1$. By the typing rules for $T\Lambda$, $\boldsymbol{T\Lambda} \vdash \mathcal{A} \rhd N_1^i : \tau_1^i$ for some $\tau_1^i \in Types$, $0 \le i \le n-1$ and $\boldsymbol{T\Lambda} \vdash \mathcal{A} \rhd M_1^j : \mu_1^j$ for some $\mu_1^j \in Types$, $1 \le j \le$. It is shown by simple induction that $\tau_1^0 = \tau$ and $\tau_1^i = \mu_1^{n-i+1} \to \tau_1^{i-1}, 1 \le i \le n$. Therefore $\mathcal{A}(x) = \tau_1^n = \mu_1^1 \to \mu_1^2 \to \cdots \to \mu_1^n \to \tau$. Since $erase(M_1) \equiv erase(M_2)$, M_2 must be of the form $(\cdots(x\ M_2^1)\cdots M_2^n)$ such that $erase(M_1^1) \equiv erase(M_2^1), \ldots, erase(M_1^n) \equiv erase(M_2^n)$. Then similarly we have $\boldsymbol{T\Lambda} \vdash \mathcal{A} \rhd M_2^j : \mu_2^j$ for some $\mu_1^i \in Types$, $0 \le j \le n-1$ and $\mathcal{A}(x) = \mu_2^1 \to \mu_2^2 \to \cdots \to \mu_2^n \to \tau$. This implies $\mu_1^2 = \mu_2^1, \ldots, \mu_1^n = \mu_2^n$. Then by induction hypothesis, $M_1^1 \equiv M_2^1, \ldots, M_1^n \equiv M_2^n$. Hence we have $M_1 \equiv M_2$.

Since M_1 is in β-normal form, we have exhausted all cases. ∎

We now prove the theorem. Let $M_1 \equiv typedterm(\Delta_1), M_2 \equiv typedterm(\Delta_2)$. Also let $\mathcal{A} \rhd M_1' : \tau, \mathcal{A} \rhd M_2' : \tau$ be terms in β-normal form such that $\mathcal{A} \rhd M_1 \twoheadrightarrow_{T\Lambda} M_1' : \tau$ and $\mathcal{A} \rhd M_2 \twoheadrightarrow_{T\Lambda} M_2' : \tau$ (such M_1', M_2' always exist). By lemma 3, there are β-normal form terms $\mathcal{A} \rhd e_1 : \tau$ and $\mathcal{A} \rhd e_2 : \tau$ such that $erase(M_1') \equiv e_1, erase(M_2') \equiv e_2$ and $\mathcal{A} \rhd e \twoheadrightarrow_{\mathrm{ML}} e_1 : \tau$ and $\mathcal{A} \rhd e \twoheadrightarrow_{\mathrm{ML}} e_2 : \tau$. By the uniqueness of normal form, $e_1 \equiv e_2$. Thus $erase(M_1') \equiv erase(M_2')$. Then by lemma 4, $M_1' \equiv M_2'$. ∎

3.3 Semantics of Core-ML

We define the semantics of Core-ML relative to a model of $T\Lambda$. We first define the semantics of Core-ML typings and then "lift" them to general Core-ML terms.

Let $\mathcal{M}$ be any given model of $T\Lambda$. The semantics of ML typings relative to $\mathcal{M}$ is defined as

$$\mathcal{M}[\![\mathcal{A} \rhd e : \tau]\!]^{\mathrm{ML}}\varepsilon = \mathcal{M}[\![\mathcal{A} \rhd typedterm(\Delta) : \tau]\!]\varepsilon$$

for some derivation Δ for $\mathcal{A} \rhd e : \tau$. By theorem 7 and the soundness of $T\Lambda$ theories (theorem 5), this definition does not depend on the choice of Δ.

For a given type assignment scheme Σ, the set of *admissible type assignments under* Σ denoted by $TA(\Sigma)$ is the set $\{\mathcal{A}|\exists\theta.\ \mathcal{A}\uparrow^{dom(\Sigma)} = \theta(\Sigma)\}$. Under a given type assignment $\mathcal{A}$, the set $TP(\mathcal{A}, \Sigma \rhd e : \rho)$ of types associated with a term $\Sigma \rhd e : \rho$ is the set $\{\tau|\exists\theta.\ (\mathcal{A}\uparrow^{dom(\Sigma)}, \tau) = \theta(\Sigma, \rho)\}$. For a set of types S, we write $\Pi\tau \in S.D_\tau$ for the space of functions f such that $dom(f) = S, f(\tau) \in D_\tau$. Then the semantics $\mathcal{M}[\![\Sigma \rhd e : \rho]\!]^{\mathrm{ML}}$ of a Core-ML term $\Sigma \rhd e : \rho$ relative to a model $\mathcal{M}$ is the function taking a type assignment $\mathcal{A} \in TA(\Sigma)$ and an environment $\varepsilon \in Env^{\mathcal{M}}(\mathcal{A})$ that returns an element in $\Pi\tau \in TP(\mathcal{A}, \Sigma \rhd e : \rho).\ D_\tau$ defined as follows:

$$\mathcal{M}[\![\Sigma \rhd e : \rho]\!]^{\mathrm{ML}}\mathcal{A}\varepsilon = \{(\tau, \mathcal{M}[\![\mathcal{A} \rhd e : \tau]\!]^{\mathrm{ML}}\varepsilon)|\tau \in TP(\mathcal{A}, \Sigma \rhd e : \rho)\}$$

For example,

$$\mathcal{M}[\![\emptyset \rhd \lambda x.\, x : t \to t]\!]^{\mathrm{ML}}\mathcal{A}\varepsilon = \{(\tau \to \tau, \mathcal{M}[\![\mathcal{A} \rhd \lambda x : \tau.\, x : \tau \to \tau]\!]\varepsilon)|\tau \in Types\}$$

Now if each element of $D_{\tau_1 \to \tau_2}$ is a function from D_{τ_1} to D_{τ_2} then by the extensionality property of $\mathcal{M}$, we have

$$\mathcal{M}[\![\emptyset \rhd \lambda x.\, x : t \to t]\!]^{\mathrm{ML}}\mathcal{A}\varepsilon = \{(\tau \to \tau, id_{D_\tau})|\tau \in Types\}$$

where id_X is the identity function on X.

4 Soundness and Completeness of Core-ML Theories

Let $\mathcal{M}$ be a given model of $T\Lambda$. $\mathcal{M}$ also determines the semantics of ML. We say that an equation $\mathcal{A} \rhd e_1 = e_2 : \rho$ is *valid* in $\mathcal{M}$, write $\mathcal{M} \models_{\mathrm{ML}} \Sigma \rhd e_1 = e_2 : \rho$, iff $\mathcal{M}[\![\Sigma \rhd e_1 : \rho]\!]^{\mathrm{ML}} = \mathcal{M}[\![\Sigma \rhd e_2 : \rho]\!]^{\mathrm{ML}}$ (as mappings). Let $\boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{M})$ be the set of all equations in Core-ML that are valid in $\mathcal{M}$. Write $\mathcal{M} \models_{\mathrm{ML}} F$ for $F \subseteq \boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{M})$.

Theorem 8 (Strong Soundness of Core-ML Theories) *Let E_{ML} be any set of ML-equations and $\mathcal{M}$ be any model. If $\mathcal{M} \models_{\mathrm{ML}} E_{\mathrm{ML}}$, then $Th_{\mathrm{ML}}(E_{\mathrm{ML}}) \subseteq \boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{M})$.*

Proof Define mappings Φ, Ψ between sets of ML-equations and sets of $T\Lambda$-equations as:

$$\begin{aligned}
\Phi(E_{\mathrm{ML}}) &= \{\mathcal{A} \rhd M = N : \tau | \text{there is some } \Sigma \rhd e_1 = e_2 : \rho \in E_{\mathrm{ML}} \text{ such that} \\
&\quad (\mathcal{A}, \tau) \text{ is an instance of } (\Sigma, \rho), erase(M) \equiv letexpd(e_1), erase(N) \equiv letexpd(e_2)\} \\
\Psi(E_{T\Lambda}) &= \{\Sigma \rhd e_1 = e_2 : \rho | \text{ for all ground instance } (\mathcal{A}, \tau) \text{ of } (\Sigma, \rho) \text{ there are some } M_1, M_2 \\
&\quad \text{such that } erase(M_1) \equiv letexpd(e_1), erase(M_2) \equiv letexpd(e_2), \mathcal{A} \rhd M_1 = M_2 : \tau \in E_{T\Lambda}\}
\end{aligned}$$

The proof uses the following lemmas.

Lemma 5 *For any set of ML-equations E_{ML}, $\Psi(Th_{T\Lambda}(\Phi(E_{\mathrm{ML}}))) = Th_{\mathrm{ML}}(E_{\mathrm{ML}})$.*

Proof By our assumptions on E_{ML} and the properties of rules of ML-theories, $\Sigma \rhd e_1 = e_2 : \rho \in Th_{\mathrm{ML}}(E_{\mathrm{ML}})$ iff for all ground instance $(\mathcal{A}, \tau)$ of (Σ, ρ), $\mathcal{A} \rhd e_1 = e_2 : \tau \in Th_{\mathrm{ML}}(E_{\mathrm{ML}})$. By definition of Ψ, $\Psi(Th_{T\Lambda}(\Phi(E_{\mathrm{ML}})))$ also has this property. It is therefore enough to show that $\mathcal{A} \rhd e_1 = e_2 : \tau \in Th_{\mathrm{ML}}(E_{\mathrm{ML}})$ iff $\mathcal{A} \rhd e_1 = e_2 : \tau \in \Psi(Th_{T\Lambda}(\Phi(E_{\mathrm{ML}})))$, which is proved by the relationship between sets of rules of $T\Lambda$ and Core-ML and the definition of Ψ. ∎

Lemma 6 *Let $\mathcal{M}$ be any model.* $\boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{M}) = \Psi(\boldsymbol{Valid}^{T\wedge}(\mathcal{M}))$.

Proof Suppose $\Sigma \rhd e_1 = e_2 : \rho \in \boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{M})$. For any ground instance $(\mathcal{A}, \tau)$ of (Σ, ρ), $\mathcal{M}[\![\mathcal{A} \rhd e_1 : \tau]\!]^{\mathrm{ML}}$ $= \mathcal{M}[\![\mathcal{A} \rhd e_2 : \tau]\!]^{\mathrm{ML}}$. Let Δ_1, Δ_2 be derivations for $\mathcal{A} \rhd e_1 : \tau$ and $\mathcal{A} \rhd e_2 : \tau$ respectively. Then $erase(typedterm(\Delta_1)) \equiv letexpd(e_1)$, $erase(typedterm(\Delta_2)) \equiv letexpd(e_2)$, and $\mathcal{M}[\![\mathcal{A} \rhd typedterm(\Delta_1) : \tau]\!] = \mathcal{M}[\![\mathcal{A} \rhd typedterm(\Delta_2) : \tau]\!]$. Therefore by definition $\Sigma \rhd e_1 = e_2 : \rho \in \Psi(\boldsymbol{Valid}^{T\wedge}(\mathcal{M}))$. Conversely, suppose $\Sigma \rhd e_1 = e_2 : \rho \in \Psi(\boldsymbol{Valid}^{T\wedge}(\mathcal{M}))$. Then by definition, for any ground instance $(\mathcal{A}, \tau)$ of (Σ, ρ), there are M, N, $\mathcal{A} \rhd M = N : \tau \in \boldsymbol{Valid}^{T\wedge}(\mathcal{M})$, $erase(M) \equiv letexpd(e_1), erase(N) \equiv letexpd(e_2)$. Then by definition $\Sigma \rhd e_1 = e_2 : \rho \in \boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{M})$. ∎

We now prove the theorem. Suppose $\mathcal{M} \models_{\mathrm{ML}} E_{\mathrm{ML}}$. By definitions of Ψ, $\mathcal{M}[\![]\!]$ and theorem 6, $\mathcal{M} \models_{T\wedge} \Phi(E_{\mathrm{ML}})$. By theorem 5, $Th_{T\wedge}(\Phi(E_{\mathrm{ML}})) \subseteq \boldsymbol{Valid}^{T\wedge}(\mathcal{M})$. Then by lemma 5 and 6, $Th_{\mathrm{ML}}(E_{\mathrm{ML}}) \subseteq \boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{M})$. ∎

Theorem 9 (Relative Completeness of Core-ML Theories) *For any set of ML-equations E_{ML} and any model $\mathcal{M}$, if $\boldsymbol{Valid}^{T\wedge}(\mathcal{M}) = Th_{T\wedge}(\Phi(E_{\mathrm{ML}}))$ then $\boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{M}) = Th_{\mathrm{ML}}(E_{\mathrm{ML}})$*

Proof By lemma 5 and 6. ∎

Then by theorem 5, we have:

Corollary 1 (Strong Completeness of Core-ML Theories) *For any ML-theory G, there exists a model $\mathcal{G}$ such that $\boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{G}) = G$.* ∎

As a special case of theorem 9, for any model $\mathcal{M}$, we have $\boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{M}) = Th_{\mathrm{ML}}(\emptyset)$ if $\boldsymbol{Valid}^{T\wedge}(\mathcal{M}) = Th_{T\wedge}(\emptyset)$. Now let $\mathcal{S}$ be a *full type structure* where D_b is a countably infinite set, $D_{\tau_1 \to \tau_2}$ is the set of all functions from D_{τ_1} to D_{τ_2} and $\bullet$ is the function application. Friedman showed that [Fri73] $\boldsymbol{Valid}^{T\wedge}(\mathcal{S}) = Th_{T\wedge}(\emptyset)$. Then we have:

Corollary 2 $\boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{S}) = Th_{\mathrm{ML}}(\emptyset)$

This means that $=_{\mathrm{ML}}$ is sound and complete in the full type structure generated by countably infinite base sets. Since $=_{\mathrm{ML}}$ is decidable, this implies that the set of all true ML equations in the full type structure is recursively enumerable.

5 Extensions of Core-ML

As a programming language, Core-ML should be extended to support recursion and various data types including recursive types. This is done by adding constants and extending the set of types and type-schemes as (possibly infinite) trees generated by various type constructor symbols.

We assume that we are given a ranked alphabet *Tycon* (always containing $\to$ of arity 2) representing a set of type constructors. As observed in [Cop85, Wan84], an appropriate class of infinite trees to support recursive types is the set of *regular trees* [Cou83]. The set of types *Types* and the set of type-schemes *Tscheme* are

extended to the set of regular trees generated by *Tycon* and $Tycon \cup Tvar$ respectively. As an example of a recursive types, the following infinite type-scheme represented by a regular system [Cou83] correspond to polymorphic list type:

$$L^t = nil + (t \times L^t)$$

where $+$ and $\times$ are binary type constructors representing sum and product and *nil* is a trivial type representing empty lists. We also assume that there is a given set of constant symbols *Const*, each of which is associated with a type-scheme. For example, the products can be introduced by assuming the following set of constants with their type-schemes:

$$\begin{aligned} pair &: t_1 \rightarrow t_2 \rightarrow (t_1 \times t_2) \\ first &: (t_1 \times t_2) \rightarrow t_1 \\ second &: (t_1 \times t_2) \rightarrow t_2 \end{aligned}$$

The set of raw terms is extended with elements of *Const* (without their associated type-schemes) and the inference system of typing is extended by the rule for constants:

(CONST) $\mathcal{A} \rhd c : \tau$ if τ is an instance of the type-scheme associated with c

We call this extended language ML. The type inference problem of ML is still decidable. Theorem 2 hold also for ML, whose proof uses the unification algorithm for regular trees by Huet [Hue76].

In order to define a semantics of the extended language, we need to extend the simply typed lambda calculus $T\Lambda$ and its semantics. The extension of the syntax of $T\Lambda$ is done simply by adding typed constants c^τ and the obvious type-checking axiom for constants. We call this extended language $T\Lambda^+$. The notion of models is extended by adding a type-preserving interpretation function $\mathcal{C}$ for constants into a frame and the condition $[\![\mathcal{A} \rhd c^\tau : \tau]\!]\varepsilon = \mathcal{C}(c^\tau)$ on the semantic mapping. Breazu-Tannen and Meyer extended [BM85] Friedman's soundness and completeness of equational theories to languages with constants and a set of types satisfying arbitrary constraints. The set of types of $T\Lambda^+$ satisfies their definition of *type algebra* and the soundness and completeness of equational theories (theorem 5) still holds for $T\Lambda^+$.

The relationship between $T\Lambda^+$ terms and derivations of ML typing-schemes is essentially unchanged and theorem 6 still holds (by adding the case for constants). However, theorem 7 no longer holds. There are non convertible $T\Lambda^+$ terms that correspond to a same ML typing. For example, consider the ML typing:

$$\emptyset \rhd (second((pair\ \lambda x.\, x)1)) : int$$

The following two $T\Lambda^+$ terms both correspond to derivations of the above typing:

$$\emptyset \rhd (second^{((int \rightarrow int) \times int) \rightarrow int}\ ((pair^{(int \rightarrow int) \rightarrow int \rightarrow ((int \rightarrow int) \times int)}\ \lambda x : int.\, x)1)) : int$$

$$\emptyset \rhd (second^{((bool \rightarrow bool) \times int) \rightarrow int}\ ((pair^{(bool \rightarrow bool) \rightarrow int \rightarrow ((bool \rightarrow bool) \times int)}\ \lambda x : bool.\, x)1)) : int$$

but they are not convertible to each other (with respect to $=_{T\Lambda}$). Another counter-example can be constructed using infinite types. An obvious implication of this fact is that we cannot interpret constants (and expressions

of infinite types) arbitrarily. We need to restrict models of $T\Lambda^+$ to those that give same denotations to terms whenever they correspond to derivations of a same ML typing. The required condition for a model $\mathcal{M}$ of $T\Lambda^+$ is that if $erase(M) \equiv erase(N)$ then $\mathcal{M}[\![\mathcal{A} \rhd M : \tau]\!] = \mathcal{M}[\![\mathcal{A} \rhd N : \tau]\!]$. We call a model $\mathcal{M}$ of $T\Lambda^+$ satisfying this condition *abstract*. By the completeness theorem for equational theories, $T\Lambda^+$ always has an abstract model. We further think that the class of abstract models covers a wide range of standard models of languages with standard set of constants. For example, ordinary interpretation of *pair* and *second* certainly satisfy the above condition and suggests an abstract model. Any abstract model of $T\Lambda^+$ yields a semantics of ML. The soundness and completeness of equational theories of ML (theorem 8 and 9) hold with respect to the class of abstract models. Proofs are same as before. The condition of abstract models can be regarded as a necessary condition for *fully abstract models* we will exploit in the next section.

6 Full Abstraction of ML

One desired property of a denotational semantics of a programming language is *full abstraction* [Mil77, Plo77, Mul84, MC88], which roughly says that the denotational semantics coincides with the operational semantics. In this section, we will show that if a model of $T\Lambda^+$ is fully abstract for an operational semantics of $T\Lambda^+$ then it is also fully abstract for the corresponding operational semantics of ML. It should be noted, however, that our notion of models of $T\Lambda^+$ assume the structure that makes the β-rule always sound. This means that a model is fully abstract for an operational semantics only if the operational semantics is faithful to the β-rule.

Following [Plo77, MC88], we define an operational semantics as a partial function on closed terms of *base types*. Let $\mathcal{E}^{T\Lambda}, \mathcal{E}^{ML}$ be respectively the evaluation functions of $T\Lambda^+$ and ML determining their operational semantics. We write $\mathcal{E}(X) \Downarrow y$ to means that $\mathcal{E}(X)$ is defined and equal to y. On the operational semantics of $T\Lambda^+$ we assume that it depend only on structure of terms. Formally, we assume $\mathcal{E}^{T\Lambda}$ to satisfy the following property:

> for two terms $\emptyset \rhd M : b$ and $\emptyset \rhd N : b$ if $erase(M) \equiv erase(N)$ then $\mathcal{E}^{T\Lambda}(\emptyset \rhd M : b) \Downarrow \emptyset \rhd c^b : b$ iff $\mathcal{E}^{T\Lambda}(\emptyset \rhd N : b) \Downarrow \emptyset \rhd c^b : b$.

We believe this condition to be satisfied by most operational semantics of explicitly-typed programming languages. On the operational semantics of $\mathcal{E}^{ML}$ we assume the following property on evaluation of *let*-expressions:

$$\mathcal{E}^{ML}(\emptyset \rhd e : b) \Downarrow \emptyset \rhd c : b \text{ iff } \mathcal{E}^{ML}(\emptyset \rhd letexpd(e) : b) \Downarrow \emptyset \rhd c : b.$$

This condition correspond to the equality axiom (*let*), which is analogous to the axiom (β). This assumption reflects our structure of models where the rule (β) is always sound. Finally we assume the following relationship between the operational semantics of $T\Lambda^+$ and the operational semantics of ML:

> for terms $\emptyset \rhd M : b$ of $T\Lambda^+$ and $\emptyset \rhd e : b$ of ML, if $erase(M) \equiv e$ then $\mathcal{E}^{T\Lambda}(\emptyset \rhd M : b) \Downarrow \emptyset \rhd c^b : b$ iff $\mathcal{E}^{ML}(\emptyset \rhd e : b) \Downarrow \emptyset \rhd c : b$.

We believe that in most cases it is routine to construct $\mathcal{E}^{ML}$ from given $\mathcal{E}^{T\Lambda}$ that satisfies the condition and vice versa.

A context $C[\,]$ in $T\Lambda^+$ is a $T\Lambda^+$ pre-term with one "hole" in it. We omit a formal definition. A context $C[\,]$ is a *closing b-context for* $\mathcal{A} \rhd M : \tau$ if there is a derivation for $\boldsymbol{T\Lambda} \vdash \emptyset \rhd C[M] : b$ such that its sub-derivation for (the occurrence in $C[M]$ of) M is a derivation for $\mathcal{A} \rhd M : \tau$. Two $T\Lambda^+$ terms $\mathcal{A} \rhd M : \tau$ and $\mathcal{A} \rhd N : \tau$ are *operationally equivalent*, write $\mathcal{A} \rhd M \overset{T\Lambda}{\approx} N : \tau$, iff for any closing b-context $C[\,]$ for these two terms, $\mathcal{E}^{T\Lambda}(\emptyset \rhd C[M] : b) \Downarrow \emptyset \rhd c^b : b$ iff $\mathcal{E}^{T\Lambda}(\emptyset \rhd C[N] : b) \Downarrow \emptyset \rhd c^b : b$.

Under our assumption on *let*-expressions, it is enough to consider raw terms and contexts that do not contain *let*-construct. Therefore we define a context $c[\,]$ in ML as a context of the untyped lambda calculus. A context $c[\,]$ is a *closing b-context for* $\Sigma \rhd e : \rho$ if there is a derivation for $\emptyset \rhd c[e] : b$ such that its subderivation for e is a derivation for an instance of $\Sigma \rhd e : \rho$. Two ML terms $\Sigma \rhd e_1 : \rho$ and $\Sigma \rhd e_2 : \rho$ are *operationally equivalent*, write $\Sigma \rhd e_1 \overset{\mathrm{ML}}{\approx} e_2 : \rho$, iff for any closing b-context $c[\,]$ for these two terms, $\mathcal{E}^{\mathrm{ML}}(\emptyset \rhd c[e_1] : b) \Downarrow \emptyset \rhd c : b$ iff $\mathcal{E}^{\mathrm{ML}}(\emptyset \rhd c[e_2] : b) \Downarrow \emptyset \rhd c : b$.

A model $\mathcal{M}$ is *fully abstract for* $\mathcal{E}^{T\Lambda}$ if $\mathcal{M} \models_{T\Lambda} \mathcal{A} \rhd M = N : \tau$ iff $\mathcal{A} \rhd M \overset{T\Lambda}{\approx} N : \tau$. Similarly, a mode $\mathcal{M}$ is *fully abstract for* $\mathcal{E}^{\mathrm{ML}}$ if $\mathcal{M} \models_{\mathrm{ML}} \Sigma \rhd e_1 = e_2 : \rho$ iff $\Sigma \rhd e_1 \overset{\mathrm{ML}}{\approx} e_2 : \rho$. Note that a model $\mathcal{M}$ is fully abstract for $\mathcal{E}^{T\Lambda}$ then it is an abstract model. This means that any fully abstract model for $\mathcal{E}^{T\Lambda}$ yields a semantics of ML. Moreover, we have:

Theorem 10 *If a model $\mathcal{M}$ is fully abstract for $\mathcal{E}^{T\Lambda}$ then $\mathcal{M}$ is also fully abstract for $\mathcal{E}^{\mathrm{ML}}$.*

Proof Let $\mathcal{M}$ be any fully abstract model for $\mathcal{E}^{T\Lambda}$. By our assumption on $\mathcal{E}^{\mathrm{ML}}$ and the definition of $\mathcal{M}[\![\,]\!]^{\mathrm{ML}}$, it is sufficient to show the condition of full abstraction for $\mathcal{E}^{\mathrm{ML}}$ for terms that do not contain *let*-construct. Suppose $\Sigma \rhd e_1 \overset{\mathrm{ML}}{\approx} e_2 : \rho$, where e_1, e_2 do not contain *let*-construct. By the definition of $\overset{\mathrm{ML}}{\approx}$, $\mathcal{A} \rhd e_1 \overset{\mathrm{ML}}{\approx} e_2 : \tau$ for any ground instance $(\mathcal{A}, \tau)$ of (Σ, ρ). Let $\mathcal{A} \rhd M : \tau, \mathcal{A} \rhd N : \tau$ be $T\Lambda^+$ terms such that $erase(M) \equiv e_1, erase(N) \equiv e_2$. By the assumption on the relationship between $\overset{T\Lambda}{\approx}$ and $\overset{\mathrm{ML}}{\approx}$, $\mathcal{A} \rhd M \overset{T\Lambda}{\approx} N : \tau$. By the full abstraction of $\mathcal{M}$ for $\mathcal{E}^{T\Lambda}$, by theorem 6 and by definition of $\mathcal{M}[\mathbf{]}^{\mathrm{ML}}$, $\mathcal{M} \models_{\mathrm{ML}} \Sigma \rhd e_1 = e_2 : \rho$. Conversely, suppose $\mathcal{M} \models_{\mathrm{ML}} \Sigma \rhd e_1 = e_2 : \rho$, where e_1, e_2 do not contain *let*-construct. Let $c[\,]$ be any closing b-context for $\Sigma \rhd e_1 : \rho$ and $\Sigma \rhd e_1 : \rho$. Let Δ_1 be a derivation for $\emptyset \rhd c[e_1] : b$ such that it contains a subderivation Δ_2 for $\mathcal{A} \rhd e_1 : \tau$ where $(\mathcal{A}, \tau)$ is an instance of (Σ, ρ). Since $c[]$ is a closing b-context for $\Sigma \rhd e_1 : \rho$, such Δ_1 always exists. Since $\Sigma \rhd e_2 : \rho$ is a term, there is a derivation tree Δ_3 for $\mathcal{A} \rhd e_2 : \tau$. Then by typing rules and the definition of contexts, the derivation tree Δ_4 obtained from Δ_1 by replacing the subtree Δ_2 by Δ_3 is a derivation for $\emptyset \rhd c[e_2] : b$. Let $M_1 \equiv typedterm(\Delta_1)$, $M_2 \equiv typedterm(\Delta_2)$, $M_3 \equiv typedterm(\Delta_3)$, $M_4 \equiv typedterm(\Delta_4)$. Then $erase(M_2) \equiv e_1$, $erase(M_3) \equiv e_2$. Clearly M_1, M_4 respectively contain M_2, M_3 as subterms. Moreover, the $T\Lambda^+$ contexts obtained from M_1, M_4 by replacing M_2, M_3 with the 'hole' are identical. Call this context $C[\,]$. Since $\mathcal{M} \models_{\mathrm{ML}} \Sigma \rhd e_1 = e_2 : \rho$, $\mathcal{M} \models_{T\Lambda} \mathcal{A} \rhd M_2 = M_3 : \tau$. Then by the full abstraction of $\mathcal{M}$ for $\mathcal{E}^{T\Lambda}$, $\mathcal{E}^{T\Lambda}(\rhd C[M_2] : \tau) \Downarrow \rhd c^b : b$ iff $\mathcal{E}^{T\Lambda}(\rhd C[M_2] : \tau) \Downarrow \rhd c^b : b$ Then by the assumption on the relationship between $\overset{T\Lambda}{\approx}$ and $\overset{\mathrm{ML}}{\approx}$, $\mathcal{E}^{\mathrm{ML}}(\emptyset \rhd c[e_1] : b) \Downarrow \emptyset \rhd c : b$ iff $\mathcal{E}^{\mathrm{ML}}(\emptyset \rhd c[e_2] : b) \Downarrow \emptyset \rhd c : b$. ∎

The importance of this result is that we can immediately apply results already developed for explicitly typed languages to implicitly typed language with ML polymorphism. As an example, Plotkin constructed a fully abstract model of his language PCF with "parallel" conditionals. It is not hard to define the "ML

version" of PCF (with parallel conditionals) by deleting type specifications of bound variables and adding let expressions. Its operational semantics can be also defined in such a way that it satisfies our assumptions. We then immediately have a fully abstract model for the ML-version of PCF.

7 Conclusion

We have presented a framework for denotational semantics and equational theories of ML. We have characterized terms of ML as representations of sets of (ground) typings. A denotation of an ML term is defined as a set of pair of a type and a value in a semantic space of the typed lambda calculus. We have axiomatized the equational theories of ML and proved their soundness and completeness with respect to our semantics. We have also shown that if a model of the typed lambda calculus is fully abstract for its operational semantics then the corresponding semantics of ML is also fully abstract for the operational semantics that correspond to the operational semantics of the typed lambda calculus. This allows us to transfer existing results of full abstraction on explicitly typed languages to ML-like languages.

One limitation of our framework is that it assumes β-equality both in equational theories and in semantic spaces (through a condition on semantic mappings). This implies that, while good for languages with "call-by-name" evaluation, our axiomatization of equational theories do not completely fit languages with "call-by-value" evaluation and our result about full abstraction is not useful for those languages. Since the "call-by-value" evaluation is a standard strategy of ML, It would be useful to develop a framework that precisely capture the equality and operational semantics under the "call-by-value" evaluation.

Acknowledgement

I would like to thank Val Breazu-Tannen and Peter Buneman for discussions and many useful suggestions.

References

[BM85] V. Breazu-Tannen and A. R. Meyer. Lambda calculus with constrained types (extended abstract). In R. Parikh, editor, *Proceedings of the Conference on Logics of Programs, Brooklyn, June 1985*, pages 23–40, *Lecture Notes in Computer Science*, Vol. 193, Springer-Verlag, 1985.

[Cop84] M. Coppo. Completeness of Type Assignment in Continuous Lambda Models. *Theoretical Computer Science*, 29:309–324, 1984.

[Cop85] M. Coppo. A Completeness Theorem for Recursively Defined Types. In G. Goos and J. Hartmanis, editors, *Automata, Languages and Programming, 12th Colloquium, LNCS 194*, pages 120–129, Splinger-Verlag, July 1985.

[Cou83] B. Courcelle. Fundamental Properties of Infinite Trees. *Theoretical Computer Science*, 25:95–169, 1983.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[Fri73] H. Friedman. Equations between functionals. In *Lecture Notes in Mathematics 453*, pages 22–33, Springer-Verlag, 1973.

[Hin69] R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. American Mathematical Society*, 146:29–60, December 1969.

[Hin83] R. Hindley. The Completeness Theorem for Typing λ-Terms. *Theoretical Computer Science*, 22:1–17, 1983.

[HMM86] R. Harper, D. B. MacQueen, and R. Milner. *Standard ML.* LFCS Report Series ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, March 1986.

[HS86] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, 1986.

[Hue76] G Huet. *Résolution d'équations dans les langages d'ordre 1,2,...ω.* PhD thesis, University Paris, 1976.

[MC88] A.R. Meyer and S.S. Cosmadakis. Semantical Paradigms. In *Proc. IEEE Symposium on Logic in Computer Science*, July 1988.

[MH88] J. C. Mitchell and R. Harper. The Essence of ML. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, pages 28–46, San Diego, California, January 1988.

[Mil77] R. Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[Mil78] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MPS86] D.B. MacQueen, G.D. Plotkin, and Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1/2):95–130, 1986.

[Mul84] K. Mulmuley. A semantic characterization of full abstraction for typed lambda calculus. In *Proc. 25-th IEEE Symposium on Fundations of Computer Science*, pages 279–288, 1984.

[Plo77] G.D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[Tur85] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science 201*, pages 1–16, Springer-Verlag, 1985.

[Wan84] M. Wand. A Types-as-Sets Semantics for Milner-Style Polymorphism. In *Proc. 11th ACM Symposium on Principles of Programming Languages*, pages 158–164, January 1984.